CS 152: Neural Networks
Final Project
Evolving a Sigma-Pi Network as a Network Simulator

Justin Basilico

December 20, 2001

1 Problem Statement

Sigma-pi networks seem to have properties that make them more powerful than a standard neural network models. This power is derived from the addition of the pi units, which take the product of their input, rather than just the sum. One way to look at these products is that it can allow for the input to the network to control the weight values in the network. In this sense, the network could be "programmed" by its input to act in a variety of different ways.

The purpose of this project is to make use of this "programmability' of sigma-pi networks to create a network that can act as a variety of other types of networks based on its input. In particular, the goal is a sigma-pi network that will act as a network simulator. It will take as input the input values for the network that it is simulating along with the values of the weights in the network it is simulating. The target output for this sigma-pi simulation network is to produce the output that the given input network (as specified by its weights) would produce on the given input value for that network. If such a network could be created, it could be a first step in looking at networks (such as sigma-pi networks) that can operate on other networks as input, similar to Turing Machines that use other Turing Machine codes as input. One specific operation that would be an interesting extension would be to create a network that could apply a learning rule to another network in order to improve it, such as backpropagation. Although such an exploration would be beyond the timeframe allotted for this project, being able to simulate one network using another network would be a first step towards this goal.

In order to create these sigma-pi simulator networks, an evolutionary approach is used. Evolutionary techniques are particularly useful for sigma-pi networks because unlike other standard models, such as backpropagation networks, the connectivity between layers is important. In particular, while for a summation unit, a weight can be zero meaning that there is no real "influence" from that unit, for a product (pi) unit if any one of its weights is zero (or close to it) then that whole unit will always have an activation of zero (or close to it). Thus for product units, there must be a distinction between when there is a connection between two nodes and when there is not. When there is a connection, then the weight can be applied. When there is not, then the activation from that unit is not passed on and not taken into account in the product. This particular problem is also seems to easily applicable for a genetic algorithm since a simulator sigma-pi network can be built that assumes a weight of 1.0 whenever there is a connection between two units. The networks will be tested against a set of randomly created networks of the specific architecture that the sigma-pi network is simulating when given random input values. Thus, the fitness of a network chromosome will is calculated as how well it simulates these random networks in terms of the average mean squared error. The goal is to evolve a chromosome that perfectly simulates these networks.

This evolutionary technique will be applied to try and evolve sigma-pi simulator networks of varying size. First they will be tested on simple networks with no hidden layers and if simulators for those networks can be evolved, a simulator for a larger network that contains a hidden layer will be attempted.

2 Approach

Since the goal is to evolve a sigma-pi network that can simulate other networks, the input to the sigma-pi network is comprised of the inputs to the network that is being simulated along with all of the weights for the network being simulated. Since sigma-pi networks have a fixed input size, this means that any one simulator network can only simulate networks of a particular architecture (or networks that would fit in the architecture but do not use all of it). The goal of the sigma-pi network is to produce the output that the given network (in terms of weights) would produce on the given input to that network.

As it turns out, a sigma-pi network can be built to do the simulation such that the weight value between any two nodes when there is a connection between them is always 1.0. This means that only the connectivity of the sigma-pi network itself needs to be created, since the weight value can be assumed to be 1.0 when there is a weight there.

Since only the connectivity of the network needs to be created, it is simple to apply a genetic algorithm to evolve a chromosome that represents the network, which is just the connectivity of the network, similar to what Miller, Todd, and Hegde (1989) did. Specifically, the chromosome just needs to be binary values of 0 or 1, which forms the connectivity matrix between each layer in the network. For instance, if there is a layer in the network of n nodes where the previous layer has m nodes (excluding the bias node), then you would need a binary string of length $n * (m + 1)$ to encode the connectivity of that layer.

The general procedure of the genetic algorithm that operated on these chromosomes for the sigma-pi networks for a population of size n was that the fitness of each chromosome was calculated (smaller fitness being better) and then the chromosomes were sorted in increasing order according to their fitness. To create the next generation, first the top 5.0% of the current generation are copied over into the next generation (elitist selection). To create the rest of the next generation, two chromosomes are selected from the current population using a rank-based selection, where the rank is based on their location in the list when ordered according to fitness. Once these two are selected, a new chromosome is created by crossing the two over, where each component of the chromosome has a 0.1 (10%) chance of being crossed over to the other chromosome. Once the crossover is done, one of the children is arbitrarily selected to go into the new population. Once these chromosomes have all been created, each one is mutated by flipping each bit in the chromosome with a probability of 0.01 (1%). This mutation rate is relatively large, however it seems to work well with using the elite and rank selection in the program.

The fitness function applied to the network is just the average mean squared error that the sigma-pi network produced by the chromosome over a testing set of 100 random networks that it is to simulate. The error is computed by comparing the actual output with the target output that is the output of the random network it is simulating. In order to simplify the task, the network being simulated is assumed to have linear activation functions instead of sigmoid functions. Sigmoid make the fitness landscape a lot harder to deal with since the output values for networks with sigmoid units will center around 0.5. This makes it so that when you use the error as the fitness function that the network will want to just guess 0.5 since it gives a pretty low average error. Using a linear activation function gives more uniformly distribution of output values, where as a sigmoid function gives closer to a normal distribution. Although most networks do use non-linear activation function ssuch as sigmoid functions, the architecture of the sigma-pi network will be the same for any activation function, so they are all assumed to be linear in order to simplify the problem. Other fitness functions were also tried on this problem such as counting the number of incorrect outputs within some threshold while using sigmoid functions. While these did well on smaller networks, but on the larger network, it did not work at all. Using the linear activation functions and using the mean squared error was the fitness function turned out to be the best combination.

The random testing networks are generated with weight values randomly assigned to a value between -5.0 and 5.0. In addition, random input values are created in the range -1.0 to 1.0. The the network is just a simple fully-connected feed-forward network that uses only summation units, as is typical in networks such

as backpropagation networks. The input to the simulator network consisted of the randomly created input values along with randomly created weight values. The target output that the simulator network is trying to achieve is the output that this random network being simulated gives on the random input values.

This procedure is applied to three different network architectures. The first architecture to be simulated is a network with two input units and one output unit. The sigma-pi network for this architecture uses five input units, three hidden units (pi), and one output unit (sigma). The second is a little larger with two input units and two output units. The sigma-pi network in this case has eight input units, six hidden units (pi), and two output units (sigma). The third network is a combination of the first two networks with two input units, one hidden layer with two units on it, and one output unit. The sigma-pi network for this architecture is larger, with eleven input units, three hidden layers of nine units (pi), five units (sigma), and three units (pi) with one output unit (sigma). After the genetic algorithm is finished on each of these tasks, the best network produced by the algorithm is then inspected and tested against another set of 100 random networks to see how good it has generalized.

3 Results

Experiment 1 For the most simple network architecture, consisting of just two input units and one output unit, a solution was evolved very quickly. In fact, it would normally take only between 10 and 25 generations to evolve a sigma-pi simulator network that had a fitness of 0.0, which means that it had zero error on simulating all of the networks from the dataset that was used to evolve the network. On the other set of networks to simulate that were not used in the evolution of the sigma-pi simulator for this simple architecture, it also had an average mean squared error of 0.0 and was 100% correct. The chromosomes for evolving this sigma-pi network only contained 22 bits, since there are only 22 possible connections between units in the sigma-pi network, which consisted of 5 input, 3 hidden, and 1 output unit. With such a small chromosome length that contains binary values, it is not surprising that the simulator for this very simple architecture was evolved. Although in general the network will evolve rather quickly, in some instances it will get stuck at a fitness of around 2.5. However, since it was so quick to evolve in most cases, this result is encouraging in that perhaps it will not be too hard to evolve sigma-pi simulators for larger networks.

During the course of creating this project, this network was always very quick to evolve as long as a reasonable fitness function was used. Adding additional features to the genetic algorithm such as the concept of groups of bits in the chromosome being crossed over together since they form a functional unit did not seem to positively or negatively effect the performance. The completely accurate simulation network also was able to evolve without using elite selection in addition to the rank selection. The crossover rate and mutation rate do effect how quickly the optimal result is evolved, but as long as the values were reasonable, it would still eventually create the solution. Early on in the project, a sigma-pi simulator was evolved for this architecture that used sigmoid units, which took only slightly more generations. However, sigmoid units were dropped, as mentioned above, because for the other networks they would tend to get stuck at local minima of just always guessing 0.5.

Experiment 2 The second experiment consists of just adding one more output node to the previous network. Although this is just a small change in the architecture of the networks being simulated, it vastly increases the size of the chromosome that must be evolved from 22 to 68, since the sigma-pi network must now have 8 input, 6 hidden, and 2 output units. Since the size of the chromosome tripled, it is not surprising that it took a lot longer to evolve a solution. In general, it would take between about 100 and 500 epochs to evolve a solution because the algorithm would could get stuck at local minima that would take many generations to get out of. Given enough generations, however, it would always be able to get out of the local minima and get a chromosome with a best fitness of 0.0, which means that the network evolved had absolutely zero error on simulating the networks in the evolution set. Like in the first experiment, this best network created with fitness 0.0 also generalized and was able to simulate all of the networks in the separate testing dataset with 100% accuracy. The local minima that this network would tend to get stuck at is around 1.34, which it could get stuck at for several hundred generations in some instance. However, after inspecting

one such network that evolved with this fitness, it seems that the network has evolved a correct solution to one of the two output units, where it is 100% correct, but it has not figured out how to simulate the other output unit. Since the network has no hidden layers, there is no interaction between the calculations for the simulation of the first and the second output units, so really the network is learning two tasks at once. In order to do this, the network is figuring out one of the tasks (one of the output units), and then figuring out the other one (the other output unit). This result is also it means that the network is evolving one part of the network that is basically the same network as in the first experiment. Thus, if this network was able to put together two versions of the network from the first experiment, perhaps the larger, multi-layer network will be able to do the same.

Since this experiment is inherently more difficult than the first one, it should not be surprising that the evolution of this network is much more sensitive to the fitness function and learning parameters used. If the fitness function is too coarse (such as evaluating the correctness within a threshold of 0.05), the genetic algorithm will get stuck in local minima a lot more often and may never converge on a solution. At first, one trick that would get the network to start off in the right direction was to add to the fitness function the number of connected units in the network (the number of bits that are 1 in the chromosome). This would push it off in the correct direction, but also created new problems where all the connections would be removed from the network so that there was not much diversity in the population. However, in general this would speed up the evolution. The fitness functions that used a threshold value for correctness rather than a straight mean squared error worked particularly well when trying to evolve a network that used sigmoid units rather than linear ones, since it overcame the problem of the local minima at 0.5. This fitness function was dropped, however, because it made it so that for the third experiment there was not enough of a difference among fitness values.

Experiment 3 Since the other two networks were not that difficult to evolve, it seemed that it would be relatively simple to evolve the larger network that simulated networks with two input, two hidden, and one output units. However, this turned out not to be the case. The sigma-pi network needed to simulate these networks that have hidden units required a total of five layers (three hidden). The input layer had 11 units, the hidden layers had 9 (pi), 5 (sigma), and 3 (pi) units, while the output layer had 1 (sigma) unit. The total length of the binary chromosome needed to encode the connectivity of this network is 180. Thus, evolving the whole connectivity of the network is a much harder problem than the previous ones. Since the networks being simulated have multiple layers, there is a large dependency between what happens in the first few layers and the last few layers. If either one is incorrect, the network will not be able to produce the correct solution. Also, since the activation values for the hidden units are not looked at by the fitness function and there is only one output, there are not different sub-problems for the network to figure out, like there is in experiment two.

Given this complexity and the size of the chromosomes, it is not surprising that the full connectivity of a simulation network for this type of network was not able to be evolved. Whenever the genetic algorithm was applied to the problem it would seem to do well at first but would then get stuck at some local minima that it would not be able to escape from even when run for tens of thousands of generations. In order to try and get the sigma-pi simulator network to evolve, multiple variations on the genetic algorithm were tried. In particular, the concept of a functional unit in the chromosome was added, where chromosomes that make up the same functional unit would be crossed over together. To do this, a functional unit consisted of all of the weights into a particular node in the network. While it seemed that adding this concept might improve performance, it did not seem to have any effect on this experiment or the two previous ones, so it was eventually taken out of the program. When this problem was first encountered, only rank selection was used, so elite selection was added as well. Adding elite selection created somewhat of a momentum in the network so that the fitness would start decreasing a lot quicker at first. However, once the algorithm reached the same local minima it would still get stuck and never improve the fitness.

Since none of these changes seemed to improve the network much, the structure of the simulator network being evolved was changed slightly by fixing part of the network so that the size of the chromosome being

evolved is smaller. Since the activation value for the output unit depends on both the weight values of the network that it is simulating and the hidden unit activation, all of these values must be available for the second-to-last layer in order to properly compute the output. However, since the weights are specified on the input layer, their values need to be passed down for two more levels unchanged. Doing this requires 3 nodes on each layer and very sparsely connected weights in order to just get these values to the proper place so they can be used. Thus, if this portion of the network is fixed so that the weight values for the second layer of weights in the network are automatically propagated through the network by fixing the connections for those units. Fixing this part of the network cuts the length of the chromosome by half, from 180 to just 90, which is much more reasonable.

For this simulation network, a solution with a fitness of 0.0 was evolved after about 3000 generations. After evolving the network, it was 100% correct on both the dataset used to evolve the network and on the other dataset that it had never seen. Although it was able to evolve a solution on some instances, on others it would not get to fitness 0.0 and would remain stuck at a local minima for thousands of generations. These local minima usually contain a parts of an optimal solution along with other parts along with other random connections that are hard to understand. Even when a complete solution is created, the algorithm will spend many hundreds of cycles stuck in one local minima before moving on to the next one. The reason it should not be too surprising that an optimal network when the network has this part of the connectivity fixed is that the remaining parts of the network only have to evolve both the networks from the previous two parts together, which is in fact just three occupancies of the smallest network. However, because of the dependency last few layers on the first few, it still is a difficult task to evolve such a network.

Future work One of the main reasons that it is probably so hard to evolve networks in this way is due to the rather coarseness of the fitness functions used. It is very hard to come up with a fitness function for these networks where there is a lot of variety in the fitness. In most cases there seem to be distinct levels in the function that correspond to local minima without much between them. A reason for this might be due to the encoding that all weights are either 1.0 or 0.0. Perhaps allowing the network to evolve the weights would smooth out the fitness landscape so that it would not spend as much time stuck in local minima. Another logical step to follow would be to try and see if the same sorts of networks can be produced using sigmoid activation units rather than linear ones, or even let those evolve along with the network. Yet another avenue to explore along with allowing the weight values to be evolved is to use a less fixed architecture for the networks. For these experiments, the size of the layers in the sigma-pi network were chosen based on outside knowledge of what numbers of units were needed for the simulator to work. It would be interesting to offer the network more flexibility in order to see if it could evolve without as many constraints and to see if it perhaps evolves different solutions to the problem.

Since the simulator network was able to be evolved, it would be interesting to explore other possibilities of having networks such as sigma-pi networks use other networks as input and do interesting things with them. As mentioned early on, one such application might be to make networks that do the delta rule for output units, then ones for hidden units, and then possibly a larger network that does backpropagation on other networks. A pi-sigma network probably would be good for doing this, given the nature of the delta rule. Another avenue of future work would be to try and evolve a network that implements a learning rule for other networks. Here the network could be tested by running networks through it multiple times and seeing if it is improving the performance of the network. Although so far the networks being simulated have been assumed to just be simple summation networks, simulators could also be built for other types of networks, such as sigma-pi networks themselves. It would also be interesting to see if any learning algorithms for sigma-pi networks could learn the same sort of simulation tasks.

Another approach would be to see if sigma-pi networks could be created to act as a variety of different types of networks based on the input without explicitly specifying the network that the sigma-pi network will have to act as. This could be seen as just programming the sigma-pi network how to act at a higher level than explicitly specifying the network itself. In any case, there is a lot of interesting exploration that

can be done in looking at these ideas, which still seem largely unexplored. The fact that networks can be simulated by other networks can form a basis for these explorations to start from.

References

[1] Geoffrey F. Miller, Peter M. Todd, and Shailesh U. Hegde. *Designing Neural Networks using Genetic Algorithms.* In J. David Schaffer, editor, Proceedings of the Third International Conference on Genetic Algorithms, pages 379-384, San Mateo, California, 1989. Philips Laboratories, Morgan Kaufman Publishers, Inc.

[2] Chalmers D.J. *The evolution of learning: An experiment in genetic connectionism.* In Touretzky D.S., Elman J.L., Sejnowski T.J., and Hinton G.E., editors, Proc. of the 1990 Connectionist Models Summer School. Mogan Kaufmann, 1990.

[3] Tony A. Plate. *Randomly connected sigma-pi neurons can form associative memories.* Network: Computation in neural systems. Vol. 11, No. 4, pages 321-332, 2000.

[4] Belew, R.K., McInerney, J., and Schraudolph, N.N. (1991). *Evolving Networks: Using the genetic algorithm with connectionist learning.* In Artificial Life II, SFI Studies in the Sciences of Complexity, vol. X, ed. C.G. Langton, C.Taylor, J.D. Farmer, & S. Rasmussen, Addison-Wesley.

[5] DJ Janson and JF Frenzel, *Training Product Unit neural networks with Genetic Algorithms*, IEEE Expert Magazine, pages 26-33, October 1993.